Breaking the memory secrecy assumption

Raoul Strackx
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
raoul.strackx@
student.kuleuven.be

Yves Younan
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
yvesy@cs.kuleuven.be

Pieter Philippaerts
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
pieter@cs.kuleuven.be

Frank Piessens
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee, Belgium
frank@cs.kuleuven.be

Sven Lachmund
DOCOMO Euro-Labs
Landsberger Strasse 312
80687 Munich, Germany
lachmund@
docomolab-euro.com

Thomas Walter
DOCOMO Euro-Labs
Landsberger Strasse 312
80687 Munich, Germany
walter@
docomolab-euro.com

ABSTRACT

Many countermeasures exist that attempt to protect against buffer overflow attacks on applications written in C and C++. The most widely deployed countermeasures rely on artificially introducing randomness in the memory image of the application. StackGuard and similar systems, for instance, will insert a random value before the return address on the stack, and Address Space Layout Randomization (ASLR) will make the location of stack and/or heap less predictable for an attacker.

A critical assumption in these probabilistic countermeasures is that attackers cannot read the contents of memory. In this paper we show that this assumption is not always justified. We identify a new class of vulnerabilities – buffer overreads – that occur in practice and that can be exploited to read parts of the memory contents of a process running a vulnerable application. We describe in detail how to exploit an application protected by both ASLR and stack canaries, if the application contains both a buffer overread and a buffer overflow vulnerability.

We also provide a detailed discussion of how this vulnerability affects other, less widely deployed probabilistic countermeasures such as memory obfuscation and instruction set randomization.

Categories and Subject Descriptors

D [**4**]: 6

General Terms

systems security

Keywords

buffer overflow, buffer overread, bypass, probabilistic countermeasure

1. INTRODUCTION

Security is an important concern for all computer users: worms and hackers have become part of everyday internet life. A particular insidious vulnerability is the buffer overflow. This vulnerability is a significant threat to the security of a system. Most of the existing buffer overflow vulnerabilities are located on the stack, and the most common way for attackers to exploit such a buffer overflow is to modify the return address of a function. By making the return address point to code they injected into the program's memory as data, they can force the program to execute any instructions with the privilege level of the program being attacked [1].

Buffer overflows still occur often in real world programs: according to the NIST's National Vulnerability Database (NVD) [25], 563 buffer overflow vulnerabilities were reported in 2008, making up 10% of the total 5,634 vulnerabilities reported in that period, only preceded by Cross Site Scripting vulnerabilities (14 %) and SQL injection vulnerabilities (19.5 %). Of those buffer overflow vulnerabilities, 436 had a high severity rating. As a result, buffer overflows make up 15% of the 2,853 vulnerabilities with a high severity rating reported in 2008, second only to SQL injection vulnerabilities (28.5 %).

Probabilistic countermeasures are countermeasures that try to prevent code injection attacks by making use of random data. Randomness can be used in a variety of ways to build countermeasures: random information can be stored at a memory location and it can later be verified to be unchanged [15, 17, 27, 23], data and code can be stored at random locations in memory [34, 6], data and code can be obfuscated by applying an XOR on the byte representation of the data or code together with a random value [5, 21, 14, 7]. This type of countermeasures makes it harder for attackers to overwrite data if it's either obfuscated or if there is a random unknown value stored in memory that can not be modified. In the case of storing data at random locations, attackers are faced with the problem that their injected code is not stored at a predictable location in memory, making it hard to direct control flow to it. A more detailed survey of probabilistic and other types of countermeasures can be found in [39, 38].

An important assumption for this type of countermeasures is that data in memory is kept secret. However this assumption is not necessarily a valid assumption, especially in programs that already contain buffer overflows. While format string vulnerabilities are often cited as a typical way of achieving information leakage, which could result in bypassing these countermeasures, this type of vulnerabilities has become rare: according to the NVD, 24 format string vulnerabilities were reported in 2008, accounting for 0.4% of all vulnerabilities found in that year.

In this paper we present a realistic vulnerability[1], which we call a *buffer overread*, that results in the memory-secrecy assumption being violated, allowing attackers to bypass such probabilistic countermeasures. We discuss a typical instantiation of this vulnerability and provide an exploit that bypasses both ASLR and Propolice on a typical computer running Linux®, even when both these countermeasures are used concurrently.

The rest of this paper is structured as follows: Section 2 briefly recaps buffer overflows and how attackers can abuse this type of vulnerabilities. Section 3 describes how probabilistic countermeasures work and discusses different types of probabilistic countermeasures and provides details on some specific implementations. Section 4 provides a proof-of-concept of a simple application and a matching exploit which bypasses both ASLR [34, 6] and Propolice [17] protection on a standard Linux® machine. Section 5 discusses if and how this vulnerability could form a problem for other types of probabilistic countermeasures. In Section 6 an overview of related work is provided, while Section 7 presents our conclusion.

2. BUFFER OVERFLOWS

Buffer overflows are the result of an out of bounds write operation on an array. In this section we briefly recap how an attacker could exploit such a buffer overflow. A detailed overview of how a buffer overflow can be exploited, can be found in [1].

Buffers can be allocated on the stack, the heap or in the data/bss section in C. For arrays that are declared in a function body, space is reserved on the stack. Buffers that are allocated dynamically (using the *malloc* function, or some other variant), are put on the heap, while arrays that are global or static are allocated in the data/bss section. The array is manipulated by means of a pointer to the first byte. Bytes within the buffer can be addressed by adding the desired index to this base pointer.

Listing 1: A C function that is vulnerable to a buffer overflow.

```
void copy(char* src, char* dst) {
  int i = 0;
  char curr = src[0];
  while(curr) {
    dst[i] = curr;
     i++;
    curr = src[i];
  }
}
```

At run-time, no information about the array size is available. Consequently, most C-compilers will generate code that will allow a program to copy data beyond the end of an array. This behavior can be used to overwrite data in the adjacent memory space. If this adjacent memory space contains data that influences control flow, an attack can be mounted. The stack usually contains control flow data: it stores the addresses to resume execution at, after a function call has completed its execution. This address is called *the return address*. Manipulating the return address might give the attacker the possibility to execute arbitrary code. In addition to the return address, the contents of other variables on the stack might also be overwritten. This could be used to manipulate the results of previous calculations or checks.

[1]A buffer overread occurred in the syslogd daemon that was shipped with FreeBSD® 3.2 [35]

Likewise, the heap also often contains important memory management information right before the allocated buffer [40]. By manipulating this information, an attacker can control the execution path of the process when the application frees the allocated memory.

Listing 1 shows a straightforward string copy function. Improper use of this function can lead to a buffer overflow, because there is no validation that the destination buffer can actually hold the input string. An attacker can thus use the buffer overflow to overwrite memory that is stored adjacent to the destination buffer.

3. PROBABILISTIC COUNTERMEASURES

Many countermeasures make use of randomness when protecting against attacks. Many different approaches exist when using randomness for protection. Canary-based countermeasures use a secret random number that is stored before an important memory location: if the random number has changed after some operations have been performed then an attack has been performed; and thus the attack is detected. Memory-obfuscation countermeasures encrypt (usually with XOR) important memory locations or other information using random numbers. Address space layout randomizers randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers encrypt the instructions while in memory and will decrypt them before execution.

These probabilistic countermeasures are widely deployed: Windows Vista contains ASLR [19] and a version of a memory obfuscator [20], while Visual Studio supports a canary-based defense since its 2002 release [8] . The Linux kernel also contains ASLR for the stack since kernel version 2.6.12 [11], while GCC has an implementation of Propolice since version 4.1 [18]. Mac OS X has also contained ASLR for libraries since Leopard [4].

3.1 Canaries

The observation that attackers usually try to overwrite the return address when exploiting a buffer overflow led to a sequence of countermeasures that were designed to protect the return address. One of the earliest examples of this type of protection is the canary-based countermeasure [15]. This type of countermeasures protects the return address by placing a value below it on the stack that must remain unchanged during program execution. Upon entering a function the canary is placed on the stack below the return address. When the function returns, the canary stored on the stack will be compared to the original canary. If the stack-stored canary has changed, an overflow has occurred and the program can be safely terminated. A canary can be a random number, or a string that is hard to replicate when exploiting a buffer overflow (e.g., a NULL byte). StackGuard [13, 15] was the first countermeasure to use canaries to offer protection against stack-based buffer overflows. However, attackers soon discovered a way of bypassing it using indirect pointer overwriting. They would overwrite a local pointer in a function and make it point to a target location. When the local pointer is dereferenced for writing, the target location is overwritten without modifying the canary [10]. Propolice [17] is an extension of StackGuard: it prevents indirect pointer overwrites by reordering the stack frame so that buffers can no longer overwrite pointers in a function. This is done by storing all buffers local to a function right below the canary, this prevents buffers from overwriting pointers local to a function.

Canaries are also used to protect other memory locations, like the management information of the memory allocator that is often overwritten using a heap-based buffer overflow [27, 23].

3.2 Memory-obfuscation

Memory-obfuscation countermeasures use an approach that is closely related to canaries: their approach is also based on random numbers. These random numbers are used to 'encrypt' specific data in memory and to decrypt it before using it in an execution. These approaches are currently used for obfuscating pointers (XOR with a secret random value) while in memory [14]. When the pointer is later used in an instruction it is first decrypted in a register (the decrypted value is never stored in memory). If an attacker attempts to overwrite the pointer with a new value, it will have the wrong value when decrypted. This will most likely cause the program to crash.

Data space randomization (DSR) encrypts the representation of data stored in memory [7]. It performs a program transformation to encrypt each value before it is stored and to decrypt it again before being used. DSR was developed to protect against non-control data attacks as well as code injection attacks.

Ignoring optimizations, this transformation can be split in three steps. First, for each data object in the program[2], a mask is generated. Second, when object a is assigned value v, the program is modified to store the outcome of the expression $v \oplus m_a$ instead, with m_a being the mask of a. Finally all expressions are transformed to decrypt the values before they are being used. While not preventing buffer overflows, these transformations will make them harder to exploit.

3.3 Address Space Layout Randomization

ASLR [34, 6] is another approach that makes executing injected code harder. Most exploits expect the memory segments to always start at a specific known address. They will attempt to overwrite the return address of a function, or some other interesting address with an address that points into their own code. However for attackers to be able to point to their own code, they must know where in memory their code resides. If the base address is generated randomly when the program is executed, it is harder for the exploit to direct the execution-flow to its injected code because it does not know the address at which the injected code is loaded.

3.4 Instruction Set Randomization

Instruction set randomization [5, 21] is another technique that can be used to prevent the injection of attacker-specified code. ISR prevents an attacker from injecting any foreign code into the application by encrypting instructions on a per process basis while they are in memory and decrypting them when they are needed for execution. Attackers are unable to guess the decryption key of the current process, so their instructions, after they've been decrypted, cause the wrong instructions to be executed. This prevents attackers from having the process execute their payload and has a large chance crashing the process due to an invalid instruction being executed.

4. BUFFER OVERREAD VULNERABILITIES

An overarching problem with probabilistic countermeasures is that they rely on memory being kept secret. However this is an assumption which relies on the absence of other particular vulnerabilities. One example of a vulnerability which breaks this assumption is a format string vulnerability. Where an attacker can print out arbitrary memory locations. However, other vulnerabilities may exist which can cause similar effects. In this section we describe a realistic vulnerability that can be used to bypass probabilistic countermeasures: *a buffer overread vulnerability*. We will also show how this type of vulnerabilities could be used by an attacker to bypass probabilistic countermeasures.

A typical way of fixing a buffer overflow vulnerability due to wrongful use of the *strcpy* function is to replace it with a call to the *strncpy* function and to provide a maximum size to be copied. For example, Listing 2 shows how a *strcpy* is correctly replaced by a *strncpy*. However, this example also demonstrates a problem that can occur when such a replacement is done: *strncpy* has slightly different semantics than *strcpy*: while *strcpy* will always ensure that the destination string is NULL-terminated, *strncpy* makes no such guarantees. If *buff* contains 100 or more characters, then *user* will not be terminated. Subsequent use of *user* in string operations could result in buffer overreads or even in buffer overflows. Such a vulnerability occurred in the syslogd daemon that was shipped with FreeBSD® 3.2 [35].

Listing 2: Any client that connects to the server will be asked a username and a password. After receiving this information in a buffer, it is copied in a separate array using the *strcpy* and *strncpy* functions. The exploit takes advantage of vulnerabilities in the use of these functions.

```
void handleConnection(int socket) {
   char user[100];
   char pass[200];
   char buff[400];
   int c = 0;

   strncpy(buff,"USER:␣",100);
   send(socket,buff,7,0);
   recv(socket,buff,400,0);
   strncpy(user,buff,100);
   snprintf(buff,400,"Hello␣%s\nPASS:",
     user);
   c = strlen(buff)+1;
   send(socket,buff,c,0);
   recv(socket,buff,400,0);
   strcpy(pass,buff);
   strncpy(buff,"Logged␣in",100);
   send(socket,buff,23,0);
}
```

Listing 2 represents a server that contains a buffer overread and a buffer overflow vulnerability that can be abused by an attacker to inject code even if this application is protected by both Propolice and ASLR. The application and the exploit were performed on an emulated (Qemu 0.9.1) ARM® processor running Debian® GNU/Linux® 4.0, but would be equally applicable to an IA32 architecture. However, the ARM architecture was chosen because of the growing number of mobile devices that contain such processors, making it a more widely used processor even than the IA32 [22].

The attack, for which we developed a working exploit, works as follows: a client process connects to the server. The server replies with a request to transmit the username. When the client sends its username to the server, it is received in the *buff*-array. Afterwards it is copied to the smaller, 100 characters long, *user*-array by calling

[2]One of the proposed optimizations only masks data objects that can be overflowed. Modification to the memory layout of the program will prevent these from causing security vulnerabilities.

the *strncpy* function[3]. This function will prevent a buffer overflow on the user variable, however, as was noted earlier, it won't always add a NULL character. To successfully execute the buffer overread, this behavior is exploited. Therefore the client will return a series of 100 or more characters. After the server receives the client's username, it constructs a personalized message to ask for the password. This is achieved by applying the *snprintf* function, which keeps copying characters of the username until a NULL character is reached. In a previous step, the use of the *strncpy* function was exploited to prevent the occurrence of such a character. As a result, the process will leak valuable information in this step.

On both the IA32 architecture and the ARM® architecture running Linux®, the stack grows down[4] by default. This causes the buffer overread to read information stored on the stack prior to the buffer. Figure 1 displays the stack layout of the *handleConnection* function. Note that the compiler placed the *user* buffer next to the canary, followed by the stack (SP) and frame pointer (FP) and the return address (LR). When the *snprintf* function prints out the user variable, it will continue reading data stored behind it and will print out the canary, stack pointer and frame pointer. Given that the code segment starts at a relatively low address in the virtual address space of the program, it will most likely contain a NULL character (at least on the ARM® architecture), causing the *snprinf* function to terminate. However, leakage of the canary and either the stack or frame pointer is sufficient to bypass both the canary-based protection and ASLR.

The next step in the attack is to use the information returned from the *snprintf* to exploit the vulnerable *strcpy*[5]. This results in an exploit where *pass* is overflowed. From Figure 1, we can conclude that 300 bytes can be used for the shellcode (pass+user), followed by the canary, an arbitrary frame and stack pointer, finally followed by the address of *pass*, which can be derived from the leaked frame or stack pointer address. When the *handleConnection* function returns, the injected shellcode will be executed.

In StackGuard, the stack may contain integers between the buffer and the canary. Since integers have a higher probability of containing a NULL byte because they often contain relatively small numbers, this could thwart the attack presented above. However since Propolice reorganizes the stack frame to prevent indirect pointer overwrites, it facilitates a buffer overread by ensuring that all buffers are stored right below the canary.

Exploitation of this vulnerability is analogous on a typical Linux® machine running on the IA32 architecture: the stack will contain the *user* buffer followed by the canary, the frame pointer and the return address. Leakage of the canary and frame pointer is sufficient information to bypass both protections.

5. DISCUSSION

Probabilistic countermeasures introduce diversity into the deployment of operating systems and applications, which can be an effective defense against simple automated attacks, like most worms. However, relying on the secrecy of memory locations may not be sufficient to stop a determined attacker. In fact, this determined attacker could, if a *buffer overread* is present, also create a worm that automatically reads out the secret information and could use this information to exploit a subsequent buffer overflow. In this section we discuss some variations on the previously described attack.

[3]Simply using the recv function (i.e. without explicitly terminating buff after the recv call) would result in a similar buffer overread vulnerabiilty.

[4]From higher addresses to lower addresses.

[5]Our use here of an *strcpy* function in the same function as that *strncpy* functions are used is a bit artificial to demonstrate the vulnerability, but it not unthinkable that a program may contain a buffer overflow in another function or due to the misuse of some other string manipulation function.

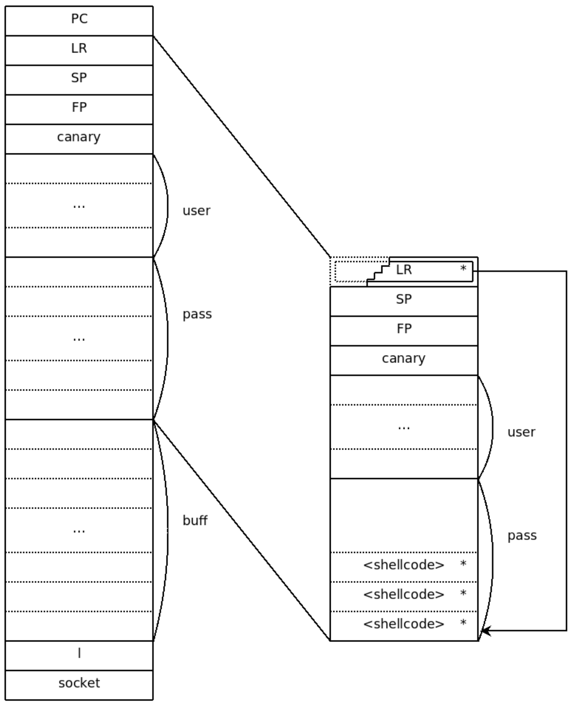

Figure 1: This figure displays the memory layout of the *handleConnection* function. It also shows the result of the attack. Memory locations marked with an "*" were updated by the exploit.

5.1 Library functions that could result in a buffer overread

While we demonstrate the vulnerability here by a misuse of the *strncpy* function, a number of other standard C library functions have similar behavior in that memory will be written to, but will not be NULL terminated by default [35]: *recv, fread, read, readv, pread, memcpy, memccpy, memmove, bcopy, gethostname, strncpy, strncat*. On top of that, manually coded array manipulation may also not correctly terminate arrays.

5.2 Partial leakage of information

If either the canary or the frame or stack pointer contain a NULL character, then the overread will terminate prematurely. However, this will also provide the attacker with information. If the canary contains a NULL byte as its last byte, the maximum amount of randomness of the canary is reduced to 2^{24}. If third is NULL, then the maximum amount of randomness is 2^{16}. If the second byte is NULL, just 256 possible canaries remain. If the first byte is NULL then the canary can be fully reconstructed. The presence of a NULL byte in the canary can however make it harder for attackers, as they must also replicate this when performing a buffer overflow. A similar approach can be applied to the available randomness of ASLR in the case the frame pointer contains a NULL byte, except that the attacker does not need to replicate the NULL character as the frame pointer does not necessarily need to be restored to its original value.

5.3 Similar attacks

A similar type of attacks to the one described in the previous section could be performed by exploiting an off-by-one overflow vulnerability to overwrite the NULL byte of a buffer that is later printed out, causing the buffer to no longer be NULL terminated. This can allow attackers to print out adjacent memory locations, they can subsequently use the information disclosed by the printed buffer to bypass the countermeasure in a subsequent vulnerability.

5.4 Other types of canaries

Two other types of canaries exist [12, 36] besides the random canary we discussed earlier: the terminator canary and the random XOR canary.

A terminator canary contains different bytes that will stop different string operations if an attacker tries to replicate them. For example, the terminator canary discussed in [12] contains the following characters: a NULL byte, a carriage return, a line feed and an EOF character. Since the terminator canary contains a NULL byte, it will terminate the buffer overread. However, not all copying operations (e.g., *memcpy*) will terminate on one of these characters. An attacker can also bypass the terminator canary by using multiple buffer overflows: the first buffer can overwrite the return address as usual, while the second overflow restores the terminator canary. As a result of these two attacks, the random canary is generally preferred.

The random XOR canary [36] will encrypt the return address with a random value and will decrypt it when used. This actually turns the canary-based countermeasure into a memory obfuscator. As such, we will discuss this type of canares together with the other memory obfuscators in Section 5.6.1.

5.5 Buffer overreads in other areas of memory

We focussed mostly on a stack-based buffer overread for our example, since canaries are currently only widely used to protect the stack and not the heap. However, the attack is just as viable on the heap. If canaries are present on the heap as in Heap-protector [27] and ContraPolice [23], an attacker that is able to perform a buffer overread can also read out the canaries.

In heap-based buffer overflows, the attacker will try to overwrite pointers stored in this memory and will use these pointers to perform an indirect pointer attack. However, the program does not always store pointers on the heap, making this approach not always applicable. A more general approach that may be used by attackers is to overwrite the memory management information that is used by memory allocators. For every chunk of memory under its control, the memory allocator under many operating systems will store management information next to it [40]. Since these memory allocators will often store pointers in the management information, they provide a reliable and portable way (i.e., not-application dependent) of performing an indirect pointer overwrite. When the modified management information is subsequently used by the memory manager, the indirect pointer overwrite is performed [30, 3].

In [27], the management information is protected by generating a checksum of the management information, encrypting (XOR) this checksum with a canary and then storing this checksum together with the management information. Before the management information is used, the checksum is calculated, encrypted and verified with the stored checksum. If the checksums do not match, the application is terminated. If attackers are able to perform a buffer overread, they can read out the encrypted checksum and since they know the other information in the chunk, generate the current checksum and extract the key. In a subsequent buffer overflow, the attacker can replace the stored checksum with one that matches the updated management information. An issue that attackers may run into when performing a buffer overread on the management information has to do with the size of the chunk. If the size of the chunk is stored in the management information before the checksum, this could stop the buffer overread. The size is stored in a 32-bit integer, however it will often be smaller than 25 bits, meaning that the size will contain a NULL byte; this could thwart the buffer overread.

In [23], a canary is stored before and after each chunk of memory. After each string operation on heap memory, a check is performed to ensure that the canary stored before the chunk still matches the canary stored after the chunk. A buffer overread could allow attackers to find out the canary and reuse it when performing a buffer overflow. The size issue is not of importance here because the canary is stored right before and after the chunk, which means that the size field is stored after the canary in memory. However since this approach stores two canaries, they do not need to be the same for each chunk. This makes the attack harder to execute: the overread must occur on either the same chunk as the overflow or the chunk must lie right behind or before it so the overread is able to read it[6]. If the overflow occurs in a chunk that lies too far from the overread, the size field again becomes an issue, because it would prevent the attacker from reading adjacent chunks.

5.6 Other probabilistic countermeasures

In Section 4 we focussed on two particular types of probabilistic countermeasures: canaries and ASLR, because they are widely deployed and implementations are readily available. However other types of probabilistic countermeasures also suffer, in varying degrees, from information leakage problems. In this section we analyze if and how buffer overread vulnerabilities impact these countermeasures. Because these countermeasures are mostly still in the research domain and no implementations are currently available, we limited our discussion to an analytical evaluation.

5.6.1 Memory-obfuscation

PointGuard.

A buffer overread combined with a buffer overflow, could allow a similar attack as described in Section 4. However in this case, the *buffer overread* would be used to read out an encrypted pointer. The attacker could then use the encrypted pointer (EP) and the original value of the pointer (P)[7], to figure out the encryption key (K), due to the following formula: $P \oplus K = EP$, which due to the nature of XOR also results in the following: $EP \oplus P = K$. Even if this countermeasure is combined with canaries, an attacker would be able to bypass the protection, assuming that return address is protected by PointGuard. ASLR combined with PointGuard and canaries could make the attack slightly harder: because the original value of the frame pointer is not known, it is not possible to directly figure out the key from the encrypted value stored on the stack. However, when multiple pointers are stored on the stack, this could simplify the attack. On ARM® Linux®, the stack contains both the frame pointer and stack pointer and since the relative distance between both is unchanged in ASLR, this allows attackers

[6]Since the canary is stored before and after each chunk, two canaries will be stored right next to each other, allowing an overread to read both.

[7]An attacker can determine the original value of the pointer from running an unprotected version of the program when developing the exploit

to figure out atleast one, possibly two (depending on the relative distance between the two pointers) bytes of the randomization, significantly reducing the actual randomization. If it is not combined with the reordering canaries, then it may be easier for attackers to get multiple pointers, with larger relative distances from the frame and stack pointers, which could allow attackers to figure out up to 3 bytes of randomization. This leaves a randomization of only 256 possibilities, which is easily bypassed by a persistent attacker.

PointGuard is a generalization of the random XOR canary discussed in 5.4, as such the random XOR canary suffers from the exact same problems.

Data Space Randomization.

By using an overread and overwrite of the same buffer, this countermeasure can be defeated. Consider following structure:

```
struct User{
    unsigned int uid;
    char name[100];
};
```

When an overread of the *name* buffer occurs, the value of *uid*, p, will be returned. However, when DSR is applied, it will encrypt the value of *uid* in memory as follows: $c = p \oplus m_{uid} \oplus m_{name}$, with m_{uid} and m_{name} being the mask of *uid* and *name* respectively. Assuming a user knows his id (p), $m_{uid} \oplus m_{name} = p \oplus c$ can be calculated. Using this key, the attacker can change value p in p' by injecting $p' \oplus m_{uid} \oplus m_{name}$ using a buffer overflow. Since the application will "encrypt" the data before storing it, the value of *uid* will be overwritten as being $p' \oplus m_{uid}$. As a result, when accessing the data stored in *uid*, p' will be read instead of p.

5.6.2 Instruction set randomization

ISR also suffers from information leakage problems: if the encrypted instructions are leaked, attackers can easily determine the encryption key in the same manner as they would for PointGuard. Subsequently they can encrypt their injected shellcode with the same key, allowing correct execution of their shellcode.

One advantage this technique has over other similar techniques is that a simple buffer overread will not allow an attacker to extract the encrypted instructions from memory. Instead, an attacker would need to overwrite a pointer to point to this memory. This pointer should then subsequently be used to provide output to the attacker. This, as well as other attacks against ISR, are described in [37].

6. RELATED WORK

The domain of circumventing countermeasures has matured together with the domain of countermeasures. In this section we examine some of the related work in the domain of circumventing probabilistic countermeasures.

6.1 Canaries

Indirect pointer overwriting [10] was an important attack in which attackers would overwrite a function local pointer and make it point to a desired location. If the pointer is subsequently dereferenced for writing with a value that is controlled by the attackers, the canary-based countermeasure could be bypassed. This led to the creation of Propolice which would reorder the local stack, thereby placing all buffers above all pointers in a stack frame, thus preventing a buffer from overwriting a function local pointer.

Richarte [26] discusses four different ways of defeating StackGuard. Three of them consist of changing the frame pointer, while the forth attack tries to change the arguments of a function. All four attacks are solved by Propolice since it also protects the frame pointer and provides protection for the arguments by copying them below the buffers and using those inside the function.

Litchfield [24] describes an attack to bypass an earlier version of the canary implementation in Visual Studio® [9]. Windows® stores the function pointers for exception handling on the stack. Attackers able to perform a buffer overflow can ignore the return address and canary and continue to write on the stack until these exception handling pointers are made to point to their injected code. Subsequently an exception is generated (e.g., a stack overflow exception or canary mismatch exception), causing the injected code to be executed.

6.2 Address space layout randomization

Shacham et al. [28] examine limitations to the amount of randomness that ASLR can use[8]. Their paper also describes a guessing attack that can be used against programs that use forking, as the forked applications are usually not rerandomized, which could allow an attacker to keep guessing by causing forks and then trying until the address is found.

In [16] a technique is discussed to bypass ASLR applied to library functions. The relative offset of the return address of the main function to an interesting libc function is calculated. Since main returns to libc, this will have the correct address range of libc. By partially overwriting this return address to change the bytes calculated by the relative offset, the attacker can now modify main's return address to perform a return-into-libc attack.

6.3 Memory-obfuscation

A problem with the approach taken by PointGuard is that XOR encryption is bytewise encryption. If an attacker only needs to overwrite one or two bytes instead of the entire pointer, then the chances of guessing the pointer correctly vastly improve (from 1 in 4 billion to 1 in 65000) [2]. If the attacker is able to control a relatively large amount of memory (e.g., with a buffer overflow), then the chances of a successful attack increase even more. While it is possible to use better encryption, it would likely be prohibitively expensive since every pointer needs to be encrypted and decrypted this way.

6.4 Instruction-set Randomization

In [33] an attack is discussed which tries to brute force the key by guessing the key one byte at a time: by using a 1 or 2 byte instruction of which can be determined if it executed correctly, the key can be brute forced. This technique requires the key to remain constant between incorrect guesses. As such, the attack must be performed on a program which is not rerandomized at start-up or which forks children that can be used in the attack, since forking a child will not rerandomize the process.

In [37], a number of attack vectors against ISR are discussed, mainly related to extracting encrypted code from memory and then using this as an attack vector as discussed in Section 5.6.2. Most other attacks in the paper discuss guessing attacks, where attackers know a number of bytes and then execute a small loader to guess the rest of the key. Another attack assumes that the attackers are able to manipulate the generation of the key[9].

[8]This limitation is due to address space limitations in 32-bit architectures: often countermeasure will limit randomness to a maximum amount of bits, which will be less than 32 bits, making guessing attacks a possibility.

[9]In the particular implementation of the ISR examined in the paper, /dev/urandom is used to generate the key. If attackers are able to redirect the file descriptor to a file descriptor they can control, they can choose the key that is used for encryption

6.5 Other related work

In [35] non-terminated adjacent memory spaces are discussed in the context of being exploitable because more data will be copied than expected. In this context it is not presented as a technique for bypassing countermeasures, but a general technique for exploiting non-typical buffer overflows.

Sotirov and Dowd [32] discuss how to bypass the probabilistic (and other) countermeasures present in Windows Vista ® by using browser exploits to bypass the countermeasures, allowing the attackers to exploit the underlying Windows Animated Cursor vulnerability[10] [31].

In a presentation by Soeder [29] related vulnerabilities that can bypass memory secrecy are discussed.

7. CONCLUSION

Probabilistic countermeasures make a number of assumptions which leave them vulnerable to attacks from different avenues.

Firstly, the memory-secrecy assumption can not be assumed to always hold (e.g., due to buffer overread or format string vulnerabilities), especially in programs which turn out to be vulnerable to buffer overflows, as these are likely to contain other bugs as well.

Secondly, XOR is a weak encryption when the original value is known or can be extrapolated. An attacker can simply combine the original value with the encrypted value and extract the encryption key, bypassing the protection entirely. Further problems also exist for XOR encryption due to its bitwise character as was demonstrated in [2]. The problem also applies to buffer overreads: if attackers can only read 1 or 2 bytes of the encypted value, they can still modify those values. Using better encryption which would make it harder to determine the key even when the encrypted value is known or which are not bitwise operations would most likely be prohibitively expensive unless hardware support for the encryption operation is provided.

Although the wide application of probabilistic countermeasures has made it harder for attackers to exploit simple buffer overflows, the authors conclude from the weaknesses inherent in these countermeasures that exploits bypassing these protections – even when they are combined – are feasible and may become increasingly common in the future.

8. REFERENCES

[1] Aleph1. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[2] Steven Alexander. Defeating compiler-level buffer overflow protection. *;login: The USENIX Magazine*, 30(3), June 2005.

[3] Anonymous. Once upon a free(). *Phrack*, 57, 2001.

[4] Apple. Mac OS X Leopard - Security Features. http://www.apple.com/macosx/features/300.html#security.

[5] Elena G. Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 281–289, Washington, D.C., U.S.A., October 2003. ACM.

[6] Sandeep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., U.S.A., August 2003. USENIX Association.

[10] Windows was vulnerable to a stack-based buffer overflow in the code it uses to parse ANI files.

[7] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, Paris, France, July 2008. Springer.

[8] Brandon Bray. Compiler security checks in depth. http://msdn.microsoft.com/en-us/library/aa290051.aspx, February 2002.

[9] Brandon Bray. Security improvements to the whidbey compiler, November 2003.

[10] Bulba and Kil3r. Bypassing Stackguard and stackshield. *Phrack*, 56, 2000.

[11] Jonathan Corbet. Address space randomization in 2.6. http://lwn.net/Articles/121845/, February 2005.

[12] Crispin Cowan. StackGuard mechanism: Emsi's vulnerability. urlhttp://www.immunix.org/StackGuard/emsi_vuln.html, November 1999.

[13] Crispin Cowan, Steve Beattie, Ryan F. Day, Calton Pu, Perry Wagle, and Eric Walthinsen. Protecting systems from stack smashing attacks with StackGuard. In *Proceedings of Linux Expo 1999*, Raleigh, North Carolina, U.S.A., May 1999.

[14] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., U.S.A., August 2003. USENIX Association.

[15] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, U.S.A., January 1998. USENIX Association.

[16] Tyler Durden. Bypassing pax aslr protection. *Phrack*, 59, July 2002.

[17] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Divison, Tokyo Research Laboratory, June 2000.

[18] Free Software Foundation. Gcc 4.1 release series – changes, new features, and fixes. http://gcc.gnu.org/gcc-4.1/changes.html, September 2007.

[19] Michael Howard. Address space layout randomization in Windows Vista. http://blogs.msdn.com/michael_howard /archive/ 2006/05/26/address-space-layout-randomization -in-windows-vista.aspx, May 2006.

[20] Michael Howard. Protecting against pointer subterfuge (kinda!). http://blogs.msdn.com/michael_howard/archive/ 2006/01/30/520200.aspx, January 2006.

[21] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, pages 272–280, Washington, D.C., U.S.A., October 2003. ACM.

[22] Tom Krazit. Armed for the living room. http://news.cnet.com/ARMed-for-the-living-room/ 2100-1006_3-6056729.html, April 2006.

[23] Andreas Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks, November 2003.

[24] David Litchfield. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server, September 2003.

[25] National Institute of Standards and Technology. National vulnerability database statistics. http://nvd.nist.gov/statistics.cfm.

[26] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection, June 2002.

[27] William Robertson, Christopher Kruegel, Darren Mutz, and Frederik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, California, U.S.A., October 2003. USENIX Association.

[28] Hovav Shacham, Matthew Page, Ben Pfaff, Eu J. Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington, D.C., U.S.A., October 2004. ACM, ACM Press.

[29] Derek Soeder. Memory retrieval vulnerabilities. http://research.eeye.com/html/papers/download/eeyeMRV-Oct2006.pdf.

[30] Solar Designer. JPEG COM marker processing vulnerability in netscape browsers. http://www.openwall.com/advisories/OW-002-netscape-jpeg.txt, July 2000.

[31] Alexander Sotirov. 0-day ani vulnerability in microsoft windows (CVE-2007-0038). http://seclists.org/bugtraq/2007/Mar/0461.htmll, March 2007.

[32] Alexander Sotirov and Mark Dowd. Bypassing browser memory protections: Setting back browser security by 10 years. In *Proceedings of BlackHat 2008*, Las Vegas, Nevada, U.S.A., August 2008.

[33] Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? the effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, U.S.A., August 2005. Usenix.

[34] The PaX Team. Documentation for the PaX project.

[35] Twitch. Taking advantage of non-terminated adjacent memory spaces. *Phrack*, 56, 2000.

[36] Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*, pages 243–256, Ottawa, Ontario, Canada, May 2003.

[37] Yoav Weiss and Elena G. Barrantes. Known/chosen key attacks against software instruction set randomization. In *22nd Annual Computer Security Applications Conference*, Miami Beach, Florida, U.S.A., December 2006. IEEE Press.

[38] Yves Younan. *Efficient Countermeasures for Software Vulnerabilities due to Memory Management Errors*. PhD thesis, Katholieke Universiteit Leuven, May 2008.

[39] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[40] Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. Security of memory allocators for C and C++. Technical Report CW419, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2005.